Lecture 5

Sparse Linear Algebra

The solution of a linear system $Ax = b$ is one of the most important computational problems in scientific computing. As we shown in the previous section, these linear systems are often derived from a set of *differential equations*, by either finite difference or finite element formulation over a discretized mesh.

The matrix A of a discretized problem is usually very *sparse*, namely it has enough zeros that can be taken advantage of algorithmically. Sparse matrices can be divided into two classes: *structured sparse matrices* and *unstructured sparse matrices*. A structured matrix is usually generated from a structured regular grid and an unstructured matrix is usually generated from a non-uniform, unstructured grid. Therefore, sparse techniques are designed in the simplest case for structured sparse matrices and in the general case for unstructured matrices.

5.1 Cyclic Reduction for Structured Sparse Linear Systems

The simplest structured linear system is perhaps the tridiagonal system of linear equations $Ax = b$ where A is symmetric and positive definite and of form

$$A = \begin{pmatrix} b_1 & c_1 & & & \\ c_1 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-2} & b_{n-1} & c_{n-1} \\ & & & c_{n-1} & b_n \end{pmatrix}$$

For example, the finite difference formulation of the one dimensional model problems

$$-u''(x) + \sigma u(x) = f(x), \qquad 0 < x < 1, \sigma \geq 0 \tag{5.1}$$

subject to the boundary conditions $u(0) = u(1) = 0$, on a uniform discretization of spacing h yields of a triangular linear system of $n = 1/h$ variables, where $b_i = 2 + \sigma h^2$ and $c_i = -1$ for all $1 \leq i \leq n$.

Sequentially, we can solve a triangular linear system $Ax = b$ by factor A into $A = LDL^T$, where D is a diagonal matrix with diagonal $(d_1, d_2, ..., d_n)$ and L is of form

$$L = \begin{pmatrix} 1 & 0 & & \\ e_1 & 1 & 0 & \\ & \ddots & \ddots & \ddots \\ & & e_{n-1} & 1 \end{pmatrix}.$$

The factorization can be computed by the following simple algorithm.

```
Algorithm Sequential Tridiagonal Solver
  1. $d_1 = b_1$
  2. $e_= c_1/d_1$
  3. for $i = 2 : n$
      (a) $d_1 = b_i - e_{i-1}c_{i-1}$
      (b) if $i < n$ then $e_i = c_i/d_i$
```

The number float point operations is $3n$ upto a additive constant. With such factorization, we can then solve the tridiagonal linear system in additional $5n$ float point operations. However, this method is very reminiscent to the naive sequential algorithm for the prefix sum whose computation graph has a critical path of length $O(n)$. The cyclic reduction, developed by Golub and Hockney [**?**], is very similar to the parallel prefix algorithm presented in Section **??** and it reduces the length of dependency in the computational graph to the smallest possible.

The basic idea of the cyclic reduction is to first eliminate the odd numbered variables to obtain a tridiagonal linear system of $\lceil n/2 \rceil$ equations. Then we solve the smaller linear system recursively. Note that each variable appears in three equations. The elimination of the odd numbered variables gives a tridiagonal system over the even numbered variables as following:

$$c'_{2i-2}x_{2i-2} + b'_{2i}x_{2i} + c'_{2i}x_{2i+2} = f'_{2i},$$

for all $2 \leq i \leq n/2$, where

$$\begin{aligned} c'_{2i-2} &= -(c_{2i-2}c_{2i-1}/b_{2i-1}) \\ b'_{2i} &= (b_{2i} - c^2_{2i-1}/b_{2i-1} - c^2 2i/b_{2i+1}) \\ c'_{2i} &= c_{2i}c_{2i+1}/b_{2i+1} \\ f'_{2i} &= f_{2i} - c_{2i-1}f_{2i-1}/b_{2i-1} - c_{2i}f_{2i+1}/b_{2i+1} \end{aligned}$$

Recursively solving this smaller linear tridiagonal system, we obtain the value of x_{2i} for all $i = 1, ..., n/2$. We can then compute the value of x_{2i-1} by the simple equation:

$$x_{2i-1} = (f_{2i-1} - c_{2i-2}x_{2i-2} - c_{2i-1}x_{2i})/b_{2i-1}.$$

By simple calculation, we can show that the total number of float point operations is equal to $16n$ upto an additive constant. So the amount of total work is doubled compare with the sequential algorithm discussed. But the length of the critical path is reduced to $O(\log n)$. It is worthwhile to point out the the total work of the parallel prefix sum algorithm also double that of the sequential algorithm. Parallel computing is about the trade-off of parallel time and the total work. The discussion show that if we have n processors, then we can solve a tridiagonal linear system in $O(\log n)$ time.

When the number of processor p is much less than n, similar to prefix sum, we hybrid the cyclic reduction with the sequential factorization algorithm. We can show that the parallel float point operations is bounded by $16n(n + \log n)/p$ and the number of round of communication is bounded by $O(\log p)$. The communication pattern is the nearest neighbor.

Cyclic Reduction has been generalized to two dimensional finite difference systems where the matrix is a block tridiagonal matrix.

5.2 Sparse Direct Methods

Direct methods for solving sparse linear systems are important because of their generality and robustness. For linear systems arising in certain applications, such as linear programming and some structural engineering applications, they are the only feasible methods for numerical factorization.

5.2.1 LU Decomposition and Gaussian Elimination

The basis of direct methods for linear system is Gaussian Elimination, a process where we zero out certain entry of the original matrix in a systematically way. Assume we want to solve $Ax = b$ where A is a sparse $n \times n$ symmetric positive definite matrix. The basic idea of the direct method is to factor A into the product of triangular matrices $A = LL^T$. Such a procedure is called *Cholesky factorization*.

The first step of the Cholesky factorization is given by the following matrix fractorization:

$$A = \begin{pmatrix} d & v^T \\ v & C \end{pmatrix} = \begin{pmatrix} \sqrt{d} & 0 \\ v/\sqrt{d} & I \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & C - (vv^T)/d \end{pmatrix} \begin{pmatrix} \sqrt{d} & v^T/\sqrt{d} \\ 0 & I \end{pmatrix}$$

where v is $n-1 \times 1$ and C is $n-1 \times n-1$. Note that d is positive since A is positive definite. The term $C - \frac{vv^t}{d}$ is the Schur complement of A. This step is called elimination and the element d is the pivot. The above decomposition is now carried out on the Schur complement recursively. We therefore have the following algorithm for the Cholesky decomposition.

```
For $k = 1, 2, \ldots, n$
    $a(k,k) = \sqrt{a(k,k)}$
    $a(k+1:n,k) = \frac{a(k+1:n,k)}{a(k,k)}$
    $a(k+1:n,k+1:n) = a(k+1:n,k+1:n) - a(k+1:n,k)^T a(k+1:n,k)$
end
```

The entries on and below the diagonal of the resulting matrix are the entries of L. The main step in the algorithm is a rank 1 update to an $n-1 \times n-1$ block.

Notice that some fill-in may occur when we carry out the decomposition. i.e., L may be significantly less sparse than A. An important problem in direct solution to sparse linear system is to find a “good” ordering of the rows and columns of the matrix to reduce the amount of fill.

As we showed in the previous section, the matrix of the linear system generated by the finite element or finite difference formulation is associated with the graph given by the mesh. In fact, the nonzero structure of each matrix A can be represented by a graph, $G(A)$, where the rows are represented by a vertex and every nonzero element by an edge. An example of a sparse matrix and its corresponding graph is given in Figure 5.1. Note that nonzero entries are marked with a symbol, whereas zero entries are not shown.

The fill-in resulting from Cholesky factorization is also illustrated in Figure 5.1. The new graph $G^+(A)$ can be computed by looping over the nodes j, in order of the row operations, and adding edges between j's higher-numbered neighbors.

In the context of parallel computation, an important parameter the height of elimination tree, which is the number of parallel elimination steps need to factor with an unlimited number of processors. The *elimination tree* defined as follows from the fill-in calculation which was described above. Let $j > k$. Define $j >_L k$ if $l_{jk} \neq 0$ where l_{jk} is the (j,k) entry of L, the result of the decomposition. Let the parent of k, $p(k)$ =min$\{j : j >_L k\}$. This defines a tree since if $\beta >_L \alpha$, $\gamma >_L \alpha$ and $\gamma > \beta$ then $\gamma >_L \beta$. The elimination tree corresponding to our matrix is shown in Figure 5.2.

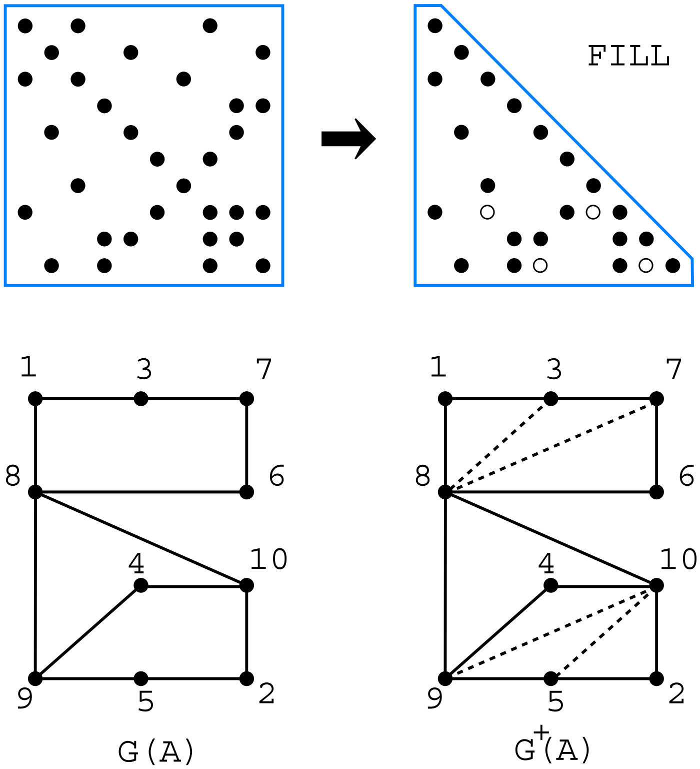


Figure 5.1: Graphical Representation of Fill-in

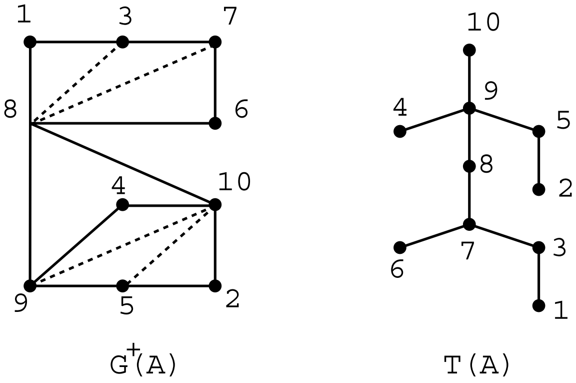


Figure 5.2: The Elimination Tree

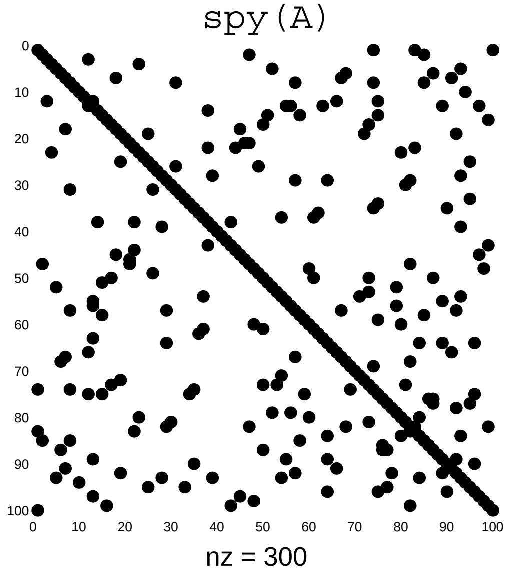


Figure 5.3: Sparsity Structure of Semi-Random Symmetric Matrix

The order of elimination determines both fill and elimination tree height. Unfortunately, but inevitably, finding the best ordering is NP-complete. Heuristics are used to reduce fill-in. The following lists some commonly used ones.

- Ordering by minimum degree (this is SYMMMD in Matlab)
- nested dissection
- Cuthill-McKee ordering.
- reverse Cuthill-McKee (SYMRCM)
- ordering by number of non-zeros (COLPERM or COLMMD)

These ordering heuristics can be investigated in Matlab on various sparse matrices. The simplest way to obtain a random sparse matrix is to use the command `A=sprand(n,m,f)`, where n and m denote the size of the matrix, and f is the fraction of nonzero elements. However, these matrices are not based on any physical system, and hence may not illustrate the effectiveness of an ordering scheme on a real world problem. An alternative is to use a database of sparse matrices, one of which is available with the command/package `ufget`.

Once we have a sparse matrix, we can view it's sparsity structure with the command `spy(A)`. An example with a randomly generated symmetric sparse matrix is given in Figure 5.3.

We now carry out Cholesky factorization of A using no ordering, and using SYMMMD. The sparsity structures of the resulting triangular matrices are given in Figure 5.4. As shown, using a heuristic-based ordering scheme results in significantly less fill in. This effect is usually more pronounced when the matrix arises from a physical problem and hence has some associated structure.

We now examine an ordering method called *nested dissection*, which uses vertex separators in a divide-and-conquer node ordering for sparse Gaussian elimination. Nested dissection [37, 38, 59] was originally a sequential algorithm, pivoting on a single element at a time, but it is an attractive

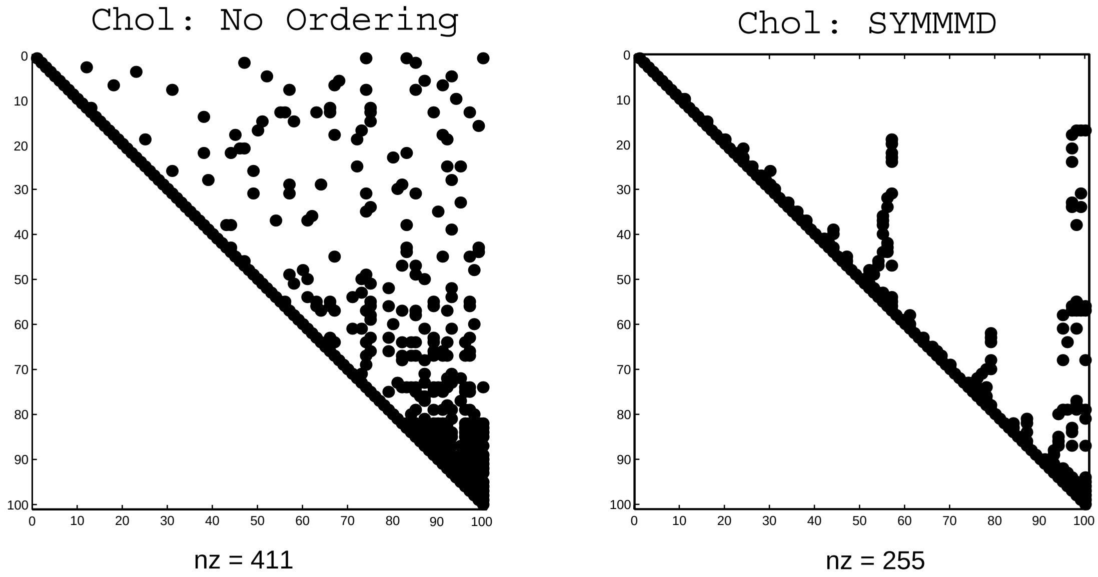


Figure 5.4: Sparsity Structure After Cholesky Factorization

parallel ordering as well because it produces blocks of pivots that can be eliminated independently in parallel [9, 29, 40, 61, 74].

Consider a regular finite difference grid. By dissecting the graph along the center lines (enclosed in dotted curves), the graph is split into four independent graphs, each of which can be solved in parallel.

The connections are included only at the end of the computation in an analogous way to domain decomposition discussed in earlier lectures. Figure 5.6 shows how a single domain can be split up into two roughly equal sized domains **A** and **B** which are independent and a smaller domain **C** that contains the connectivity.

One can now recursively order **A** and **B**, before finally proceeding to **C**. More generally, begin by recursively ordering at the leaf level and then continue up the tree. The question now arises as to how much fill is generated in this process. A recursion formula for the fill F generated for such

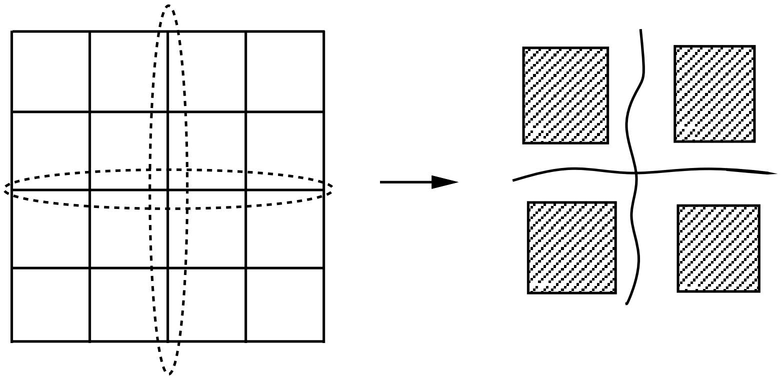


Figure 5.5: Nested Dissection

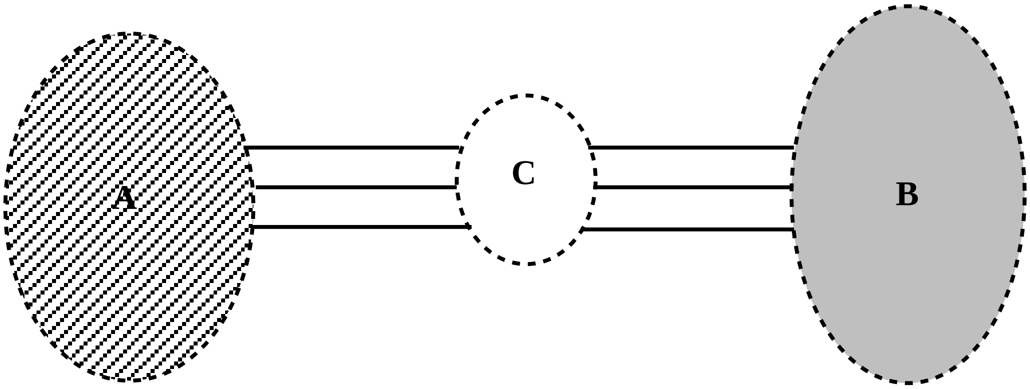


Figure 5.6: Vertex Separators

a 2-dimension nested dissection algorithm is readily derived.

$$F(n) = 4F(\frac{n}{2}) + \frac{(2\sqrt{n})^2}{2} \tag{5.2}$$

This yields upon solution

$$F(n) = 2n\log(n) \tag{5.3}$$

In an analogous manner, the elimination tree height is given by:

$$H(n) = H(\frac{n}{2}) + 2\sqrt{n} \tag{5.4}$$

$$H(n) = const \times \sqrt{n} \tag{5.5}$$

Nested dissection can be generalized to three dimensional regular grid or other classes of graphs that have small separators. We will come back to this point in the section of graph partitioning.

5.2.2 Parallel Factorization: the Multifrontal Algorithm

Nested dissection and other heuristics give the ordering. To factor in parallel, we need not only find a good ordering in parallel, but also to perform the elimination in parallel. To achieve better parallelism and scalability in elimination, a popular approach is to modify the algorithm so that we are performing a rank k update to an $n-k \times n-k$ block. The basic step will now be given by

$$A = \begin{pmatrix} D & V^T \\ V & C \end{pmatrix} = \begin{pmatrix} L_D & 0 \\ VL_D^{-T} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - VD^{-1}V^T \end{pmatrix} \begin{pmatrix} L_D^T & L_D^{-1}V^T \\ 0 & I \end{pmatrix}$$

where C is $n-k \times n-k$, V is $n-k \times k$ and $D = L_D L_D^T$ is $k \times k$. D can be written in this way since A is positive definite. Note that $VD^{-1}V^T = (VL_D^{-T})(L_D^{-1}V^T)$.

The elimination tree shows where there is parallelism since we can "go up the separate branches in parallel." i.e. We can update a column of the matrix using only the columns below it in the elimination tree. This leads to the multifrontal algorithm. The sequential version of the algorithm is given below. For every column j there is a block $\bar{U_j}$ (which is equivalent to $VD^{-1}V^T$).

$$\bar{U_j} = -\sum_k \begin{pmatrix} l_{jk} \\ l_{i_1k} \\ \vdots \\ l_{i_rk} \end{pmatrix} \left(l_{jk}\ l_{i_1k}\ \ldots\ l_{i_rk} \right)$$

where the sum is taken over all descendants of j in the elimination tree. $j, i_1, i_2, \ldots, i_r$ are the indices of the non-zeros in column j of the Cholesky factor.

For $j = 1, 2, \ldots, n$. Let $j, i_1, i_2, \ldots, i_r$ be the indices of the non-zeros in column j of L. Let $c_1, \ldots, c_s$ be the children of j in the elimination tree. Let $\bar{U} = U_{c_1} \updownarrow \ldots \updownarrow U_{c_s}$ where the U_i's were defined in a previous step of the algorithm. $\updownarrow$ is the extend-add operator which is best explained by example. Let

$$R = \begin{array}{c} \\ 5 \\ 8 \end{array} \!\!\begin{array}{c} \begin{array}{cc} 5 & 8 \end{array} \\ \begin{pmatrix} p & q \\ u & v \end{pmatrix} \end{array}, S = \begin{array}{c} \\ 5 \\ 9 \end{array} \!\!\begin{array}{c} \begin{array}{cc} 5 & 9 \end{array} \\ \begin{pmatrix} w & x \\ y & z \end{pmatrix} \end{array}$$

(The rows of R correspond to rows 5 and 8 of the original matrix etc.) Then

$$R \updownarrow S = \begin{array}{c} \\ 5 \\ 8 \\ 9 \end{array} \!\!\begin{array}{c} \begin{array}{ccc} 5 & 8 & 9 \end{array} \\ \begin{pmatrix} p+w & q & x \\ u & v & 0 \\ y & 0 & z \end{pmatrix} \end{array}$$

Define

$$F_j = \begin{pmatrix} a_{jj} & \ldots & a_{ji_r} \\ \vdots & \ddots & \\ a_{i_rj} & \ldots & a_{i_ri_r} \end{pmatrix} \updownarrow \bar{U}$$

(This corresponds to $C - VD^{-1}V^T$)

Now factor F_j

$$\begin{pmatrix} l_{jj} & 0 & \ldots & 0 \\ l_{i_1j} & & & \\ \vdots & & I & \\ l_{i_rj} & & & \end{pmatrix} \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & & & \\ \vdots & & U_j & \\ 0 & & & \end{pmatrix} \begin{pmatrix} l_{jj} & l_{i_1j} & \ldots & l_{i_rj} \\ 0 & & & \\ \vdots & & I & \\ 0 & & & \end{pmatrix}$$

(Note that U_j has now been defined.)

We can use various BLAS kernels to carry out this algorithm. Recently, Kumar and Karypis have shown that direct solver can be parallelized efficiently. They have designed a parallel algorithm for factorization of sparse matrices that is more scalable than any other known algorithm for this problem. They have shown that our parallel Cholesky factorization algorithm is asymptotically as scalable as any parallel formulation of dense matrix factorization on both mesh and hypercube architectures. Furthermore, their algorithm is equally scalable for sparse matrices arising from two- and three-dimensional finite element problems.

They have also implemented and experimentally evaluated the algorithm on a 1024-processor nCUBE 2 parallel computer and a 1024-processor Cray T3D on a variety of problems. In structural engineering problems (Boeing-Harwell set) and matrices arising in linear programming (NETLIB set), the preliminary implementation is able to achieve 14 to 20 GFlops on a 1024-processor Cray T3D.

In its current form, the algorithm is applicable only to Cholesky factorization of sparse symmetric positive definite (SPD) matrices. SPD systems occur frequently in scientific applications and are the most benign in terms of ease of solution by both direct and iterative methods. However, there are many applications that involve solving large sparse linear systems which are not SPD. An efficient parallel algorithm for a direct solution to non-SPD sparse linear systems will be extremely valuable because the theory of iterative methods is far less developed for general sparse linear systems than it is for SPD systems.

5.3 Basic Iterative Methods

These methods will focus on the solution to the linear system $Ax = b$ where $A \in \mathcal{R}^{n \times n}$ and $x, b \in \mathcal{R}^n$, although the theory is equally valid for systems with complex elements.

The basic outline of the iterative methods is as follows: Choose some initial guess, x_o, for the solution vector. Generate a series of solution vectors, $\{x_1, x_2, \ldots, x_k\}$, through an iterative process taking advantage of previous solution vectors.

Define x^* as the true (optimal) solution vector. Each iterative solution vector is chosen such that the absolute error, $e_i = \|x^* - x_i\|$, is decreasing with each iteration for some defined norm. Define also the residual error, $r_i = \|b - Ax_i\|$, at each iteration. These error quantities are clearly related by a simple transformation through A.

$$r_i = b - Ax_i = Ax^* - Ax_i = Ae_i$$

5.3.1 SuperLU-dist

SuperLU-dist is an iterative and approximate method for solving $Ax = b$. This simple algorithm eliminates the need for pivoting. The elimination of pivoting enhances parallel implementations due to the high communications overhead that pivoting imposes. The basic SuperLU-dist algorithm is as follows:

Algorithm: SuperLU-dist

```
1. $r = b - A * x$
2. $backerr = max_i(\frac{r_i}{(|A|*|x|+|b|)_i})$
3. if $(backerr < \epsilon)$ or $(backerr > \frac{lasterr}{2})$ then stop
4. solve: $L * U * dx = r$
5. $x = x + dx$
6. $lasterr = backerr$
7. loop to step 1
```

In this algorithm, x, L, and U are approximate while r is exact. This procedure usually converges to a reasonable solution after only 0-3 iterations and the error is on the order of 10^{-n} after n iterations.

5.3.2 Jacobi Method

Perform the matrix decomposition $A = D - L - U$ where D is some diagonal matrix, L is some strictly lower triangular matrix and U is some strictly upper triangular matrix.

Any solution satisfying $Ax = b$ is also a solution satisfying $Dx = (L + U)x + b$. This presents a straightforward iteration scheme with small computation cost at each iteration. Solving for the solution vector on the right-hand side involves inversion of a diagonal matrix. Assuming this inverse exists, the following iterative method may be used.

$$x_i = D^{-1}\left(L + U\right)x_{i-1} + D^{-1}b$$

This method presents some nice computational features. The inverse term involves only the diagonal matrix, D. The computational cost of computing this inverse is minimal. Additionally, this may be carried out easily in parallel since each entry in the inverse does not depend on any other entry.

5.3.3 Gauss-Seidel Method

This method is similar to Jacobi Method in that any solution satisfying $Ax = b$ is now a solution satisfying $(D - L)x = Ub$ for the $A = D - L - U$ decomposition. Assuming an inverse exists, the following iterative method may be used.

$$x_i = (D - L)^{-1}Ux_{i-1} + (D - L)^{-1}$$

This method is often stable in practice but is less easy to parallelize. The inverse term is now a lower triangular matrix which presents a bottleneck for parallel operations.

This method presents some practical improvements over the Jacobi method. Consider the computation of the j^{th} element of the solution vector x_i at the i^{th} iteration. The lower triangular nature of the inverse term demonstrates only the information of the $(j + 1)^{th}$ element through the n^{th} elements of the previous iteration solution vector x_{i-1} are used. These elements contain information not available when the j^{th} element of x_{i-1} was computed. In essence, this method updates using only the most recent information.

5.3.4 Splitting Matrix Method

The previous methods are specialized cases of Splitting Matrix algorithms. These algorithms utilize a decomposition $A = M - N$ for solving the linear system $Ax = b$. The following iterative procedure is used to compute the solution vector at the i^{th} iteration.

$$Mx_i = Nx_{i-1} + b$$

Consider the computational tradeoffs when choosing the decomposition.

- cost of computing M^{-1}
- stability and convergence rate

It is interesting the analyze convergence properties of these methods. Consider the definitions of absolute error, $e_i = x^* - x_i$, and residual error, $r_i = Ax_i - b$. An iteration using the above algorithm yields the following.

$$\begin{aligned} x_1 &= M^{-1}Nx_0 + M^{-1}b \\ &= M^{-1}(M-A)x_0 + M^{-1}b \\ &= x_0 + M^{-1}r_0 \end{aligned}$$

A similar form results from considering the absolute error.

$$\begin{aligned} x^* &= x_0 + e_0 \\ &= x_0 + A^{-1}r_0 \end{aligned}$$

This shows that the convergence of the algorithm is in some way improved if the M^{-1} term approximates A^{-1} with some accuracy. Consider the amount of change in the absolute error after this iteration.

$$\begin{aligned} e_1 &= A^{-1}r_0 - M^{-1}r_0 \\ &= e_0 - M^{-1}Ae_0 \\ &= M^{-1}Ne_0 \end{aligned}$$

Evaluating this change for a general iteration shows the error propagation.

$$e_i = \left(M^{-1}N\right)^i e_0$$

This relationship shows a bound on the error convergence. The largest eigenvalue, or spectral eigenvalue, of the matrix $M^{-1}N$ determines the rate of convergence of these methods. This analysis is similar to the solution of a general difference equation of the form $x_k = Ax_{k-1}$. In either case, the spectral radius of the matrix term must be less than 1 to ensure stability. The method will converge to 0 faster if all the eigenvalue are clustered near the origin.

5.3.5 Weighted Splitting Matrix Method

The splitting matrix algorithm may be modified by including some scalar weighting term. This scalar may be likened to the free scalar parameter used in practical implementations of Newton's method and Steepest Descent algorithms for optimization programming. Choose some scalar, w, such that $0 < w < 1$, for the following iteration.

$$\begin{aligned} x_i &= (1-w)x_0 + w\left(x_0 + M^{-1}v_0\right) \\ &= x_0 + wM^{-1}v_0 \end{aligned}$$

5.4 Red-Black Ordering for parallel Implementation

The concept of ordering seeks to separate the nodes of a given domain into subdomains. Red-black ordering is a straightforward way to achieve this. The basic concept is to alternate assigning a "color" to each node. Consider the one- and two-dimensional examples on regular grids..

The iterative procedure for these types of coloring schemes solves for variables at nodes with a certain color, then solves for variables at nodes of the other color. A linear system can be formed with a block structure corresponding to the color scheme.

$$\begin{bmatrix} \text{BLACK} & \text{MIXED} \\ \text{MIXED} & \text{RED} \end{bmatrix}$$

This method can easily be extended to include more colors. A common practice is to choose colors such that no nodes has neighbors of the same color. It is desired in such cases to minimize the number of colors so as to reduce the number of iteration steps.

5.5 Conjugate Gradient Method

The Conjugate Gradient Method is the most prominent iterative method for solving sparse symmetric positive definite linear systems. We now examine this method from parallel computing perspective. The following is a copy of a pseudocode for the conjugate gradient algorithm.
Algorithm: Conjugate Gradient

```
1. $\underline{x_0} = 0, \quad r_0 = b - A\underline{x_0} = b$
2. do $m = 1$, to $n$ steps
    (a) if $m = 1$, then $p_1 = r_0$
        else
        $\beta = r_{m-1}^T r_{m-1} / r_{m-2}^T r_{m-2}$
        $p_m = r_{m-1} + \beta p_{m-1}$
        endif
    (b) $\alpha_m = r_{m-1}^T r_{m-1} / p_m^T A p_m$
    (c) $x_m = x_{m-1} + \alpha_m p_m$
    (d) $r_m = r_{m-1} - \alpha_m A p_m$
```

When A is symmetric positive definite, the solution of $Ax = b$ is equivalent to find a solution to the following quadratic minimization problem.

$$\min_{\underline{x}} \phi(x) = \frac{1}{2} x^T A x - x^T b.$$

In this setting, $r_0 = -\nabla\phi$ and $p_i^T A p_j = 0$, i.e., p_i^T and p_j are *conjugate* with respect to A.
How many iterations shall we perform and how to reduce the number of iterations?

Theorem 5.5.1 *Suppose the condition number is $\kappa(A) = \lambda_{max}(A)/\lambda_{min}(A)$, since A is Symmetric Positive Definite, $\forall x_0$, suppose x^* is a solution to $Ax = b$, then*

$$||x^* - x_m||_A \leq 2||x^* - x_0||_A (\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1})^m,$$

where $||V||_A = V^T A V$

Therefore, $||e_m|| \leq 2||e_0|| \cdot (\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})^m$.

Another high order iterative method is Chebyshev iterative method. We refer interested readers to the book by Own Axelsson (Iterative Solution Methods, Cambridge University Press). Conjugate gradient method is a special Krylov subspace method. Other examples of Krylov subspace are GMRES (Generalized Minimum Residual Method) and Lanczos Methods.

5.5.1 Parallel Conjugate Gradient

Within each iteration of the conjugate gradient algorithm a single matrix-vector product must be taken. This calculation represents a bottleneck and the performance of conjugate gradient can be improved by parallelizing this step.

First, the matrix (A), vector (x), and solution vector (y) are laid out by rows across multiple processors as shown in Figure 5.7.

The algorithm for the distributed calculation is then simple: On each processor j, broadcast $x(j)$ and then compute $y(j) = A(j, :) * x$.

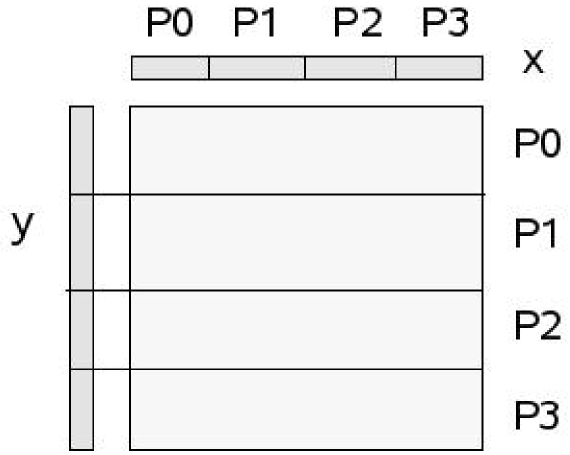


Figure 5.7: Example distribution of A, x, and b on four processors

5.6 Preconditioning

Preconditioning is important in reducing the number of iterations needed to converge in many iterative methods. Put more precisely, preconditioning makes iterative methods possible in practice. Given a linear system $Ax = b$ a parallel preconditioner is an invertible matrix C satisfying the following:

1. The inverse C^{-1} is relatively easy to compute. More precisely, after preprocessing the matrix C, solving the linear system $Cy = b'$ is much easier than solving the system $Ax = b$. Further, there are fast parallel solvers for $Cy = b'$.

2. Iterative methods for solving the system $C^{-1}Ax = C^{-1}b$, such as, conjugate gradient[1] should converge much more quickly than they would for the system $Ax = b$.

Generally a preconditioner is intended to reduce $\kappa(A)$.

Now the question is: *how to choose a preconditioner C?* There is no definite answer to this. We list some of the popularly used preconditioning methods.

- The basic splitting matrix method and SOR can be viewed as preconditioning methods.

- **Incomplete factorization preconditioning**: the basic idea is to first choose a good "sparsity pattern" and perform factorization by Gaussian elimination. The method rejects those fill-in entries that are either small enough (relative to diagonal entries) or in position outside the sparsity pattern. In other words, we perform an approximate factorization L^*U^* and use this product as a preconditioner. One effective variant is to perform block incomplete factorization to obtain a preconditioner.

  The incomplete factorization methods are often effective when tuned for a particular application. The methods also suffer from being too highly problem-dependent and the condition number usually improves by only a constant factor.

[1] In general the matrix $C^{-1}A$ is not symmetric. Thus the formal analysis uses the matrix LAL^T where $C^{-1} = LL^T$ [?].

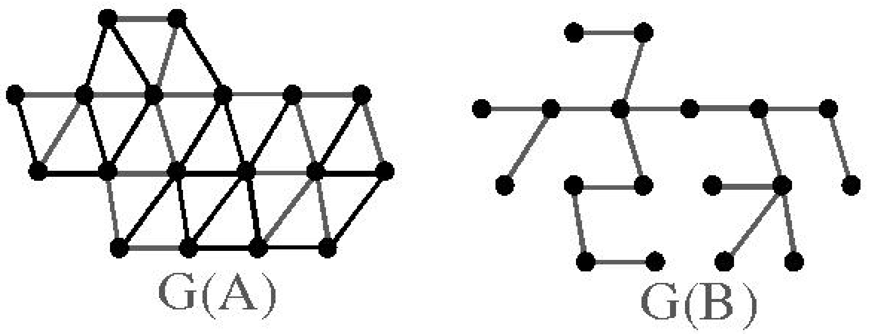


Figure 5.8: Example conversion of the graph of matrix A (G(A)) to a subgraph (G(B))

- **Subgraph preconditioning**: The basic idea is to choose a subgraph of the graph defined by the matrix of the linear system so that the linear system defined by the subgraph can be solved efficiently and the edges of the original graph can be embedded in the subgraph with small congestion and dilation, which implies small condition number of the preconditioned matrix. In other words, the subgraph can "support" the original graph. An example of converting a graph to a subgraph is shown in Figure 5.8.

  The subgraph can be factored in $O(n)$ space and time and applying the preconditioner takes $O(n)$ time per iteration.

- **Block diagonal preconditioning**: The observation of this method is that a matrix in many applications can be naturally partitioned in the form of a 2 × 2 blocks

  $$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

  Moreover, the linear system defined by A_{11} can be solved more efficiently. Block diagonal preconditioning chooses a preconditioner with format

  $$C = \begin{pmatrix} B_{11} & 0 \\ 0 & B_{22} \end{pmatrix}$$

  with the condition that B_{11} and B_{22} are symmetric and

  $$\alpha_1 A_{11} \leq B_{11} \leq \alpha_2 A_{11}$$
  $$\beta_1 A_{22} \leq B_{22} \leq \beta_2 A_{22}$$

  Block diagonal preconditioning methods are often used in conjunction with domain decomposition technique. We can generalize the 2-block formula to multi-blocks, which correspond to multi-region partition in the domain decomposition.

- **Sparse approximate inverses**: Sparse approximate inverses (B^{-1}) of A can be computed such that $A \approx B^{-1}$. This inverse is computed explicitly and the quantity $||B^{-1}A - I||_F$ is minimized in parallel (by columns). This value of B^{-1} can then be used as a preconditioner. This method has the advantage of being very parallel, but suffers from poor effectiveness in some situations.

5.7 Symmetric Supernodes

The following Section on Symmetric Supernodes is an edited excerpt from "A Supernodal Approach to Sparse Partial Pivoting," by Demmel, Eisenstat, Gilbert, Li, and Liu.

The idea of a supernode is to group together columns with the same nonzero structure, so they can be treated as a dense matrix for storage and computation. In the factorization $A = LL^T$ (or $A = LDL^T$), a supernode is a range $(r : s)$ of columns of L with the same nonzero structure below the diagonal; that is, $L(r : s, r : s)$ is full lower triangular and every row of $L(s : n, r : s)$ is either full or zero.

All the updates from columns of a supernode are summed into a dense vector before the sparse update is performed. This reduces indirect addressing and allows the inner loops to be unrolled. In effect, a sequence of col-col updates is replaced by a supernode-column (sup-col) update. The sup-col update can be implemented using a call to a standard dense Level 2 BLAS matrix-vector multiplication kernel. This idea can be further extended to supernode-supernode (sup-sup) updates, which can be implemented using a Level 3 BLAS dense matrix-matrix kernel. This can reduce memory traffic by an order of magnitude, because a supernode in the cache can participate in multiple column updates. Ng and Peyton reported that a sparse Cholesky algorithm based on sup-sup updates typically runs 2.5 to 4.5 times as fast as a col-col algorithm.

To sum up, supernodes as the source of updates help because of the following:

1. The inner loop (over rows) has no indirect addressing. (Sparse Level 1 BLAS is replaced by dense Level 1 BLAS.)

2. The outer loop (over columns in the supernode) can be unrolled to save memory references. (Level 1 BLAS is replaced by Level 2 BLAS.)

   Supernodes as the destination of updates help because of the following:

3. Elements of the source supernode can be reused in multiple columns of the destination supernode to reduce cache misses. (Level 2 BLAS is replaced by Level 3 BLAS.)

Supernodes in sparse Cholesky can be determined during symbolic factorization, before the numeric factorization begins. However, in sparse LU, the nonzero structure cannot be predicted before numeric factorization, so we must identify supernodes on the fly. Furthermore, since the factors L and U are no longer transposes of each other, we must generalize the definition of a supernode.

5.7.1 Unsymmetric Supernodes

There are several possible ways to generalize the symmetric definition of supernodes to unsymmetric factorization. We define $F = L + U - I$ to be the *filled matrix* containing both L and U.

T1 Same row and column structures: A supernode is a range $(r : s)$ of columns of L and rows of U, such that the diagonal block $F(r : s, r : s)$ is full, and outside that block all the columns of L in the range have the same structure and all the rows of U in the range have the same structure. T1 supernodes make it possible to do sup-sup updates, realizing all three benefits.

T2 Same column structure in L: A supernode is a range $(r : s)$ of columns of L with triangular diagonal block full and the same structure below the diagonal block. T2 supernodes allow sup-col updates, realizing the first two benefits.

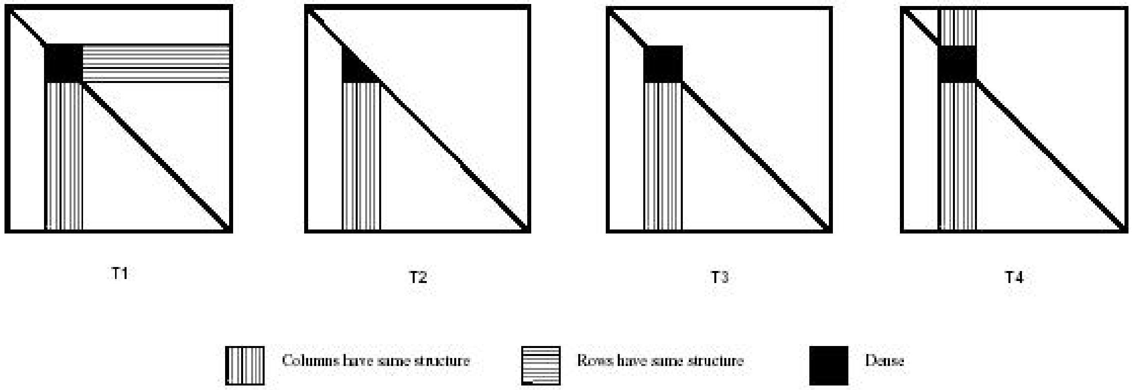


Figure 5.9: Four possible types of unsymmetric supernodes.

T3 Same column structure in L, full diagonal block in U: A supernode is a range $(r : s)$ of columns of L and U, such that the diagonal block $F(r : s, r : s)$ is full, and below the diagonal block the columns of L have the same structure. T3 supernodes allow sup-col updates, like T2. In addition, if the storage for a supernode is organized as for a two-dimensional (2-D) array (for Level 2 or 3 BLAS calls), T3 supernodes do not waste any space in the diagonal block of U.

T4 Same column structure in L and U: A supernode is a range $(r : s)$ of columns of L and U with identical structure. (Since the diagonal is nonzero, the diagonal block must be full.) T4 supernodes allow sup-col updates, and also simplify storage of L and U.

T5 Supernodes of A^TA: A supernode is a range $(r : s)$ of columns of L corresponding to a Cholesky supernode of the symmetric matrix A^TA. T5 supernodes are motivated by the observation that (with suitable representations) the structures of L and U in the unsymmetric factorization $PA = LU$ are contained in the structure of the Cholesky factor of A^TA. In unsymmetric LU, these supernodes themselves are sparse, so we would waste time and space operating on them. Thus we do not consider them further.

Figure 5.9 is a schematic of definitions T1 through T4.

Supernodes are only useful if they actually occur in practice. We reject T4 supernodes as being too rare to make up for the simplicity of their storage scheme. T1 supernodes allow Level 3 BLAS updates, but we can get most of their cache advantage with the more common T2 or T3 supernodes by using supernode-panel updates. Thus we conclude that either T2 or T3 is best by our criteria.

Figure 5.10 shows a sample matrix and the nonzero structure of its factors with no pivoting. Using definition T2, this matrix has four supernodes: {1, 2}, {3}, {4, 5, 6}, and {7, 8, 9, 10}. For example, in columns 4, 5, and 6 the diagonal blocks of L and U are full, and the columns of L all have nonzeros in rows 8 and 9. By definition T3, the matrix has five supernodes: {1, 2}, {3}, {4, 5, 6}, {7}, and {8, 9, 10}. Column 7 fails to join {8, 9, 10} as a T3 supernode because u_{78} is zero.

5.7.2 The Column Elimination Tree

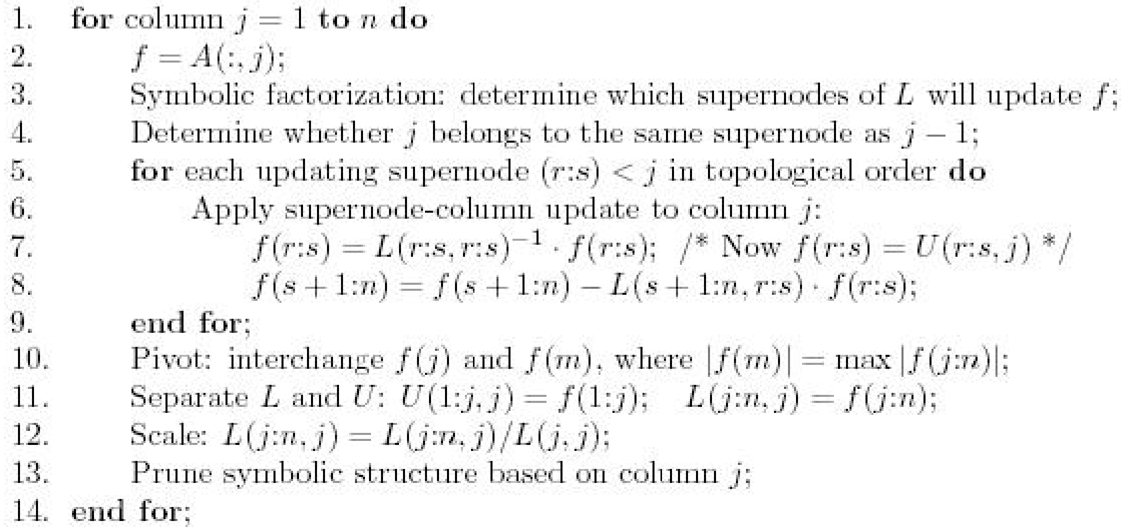
Since our definition requires the columns of a supernode to be contiguous, we should get larger supernodes if we bring together columns of L with the same nonzero structure. But the column ordering is fixed, for sparsity, before numeric factorization; what can we do?

$$\underset{\text{Original matrix } A}{\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & & & & \\
\bullet & 2 & & \bullet & & & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & \bullet & & & \bullet & \\
& & & \bullet & & & \bullet & & & \\
\bullet & & & & & 6 & & & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & & \bullet & & & & & & & \\
& & & \bullet & & \bullet & & & 9 & \\
& & \bullet & & & & \bullet & & \bullet & 10
\end{pmatrix}}
\qquad
\underset{\text{Factors } F = L + U - I}{\begin{pmatrix}
1 & \bullet & \bullet & & & \bullet & & & & \\
\bullet & 2 & \bullet & \bullet & & \bullet & & \bullet & & \\
& & 3 & & & & \bullet & \bullet & & \\
& & & 4 & \bullet & \bullet & & & \bullet & \\
& & & \bullet & 5 & \bullet & \bullet & & \bullet & \\
\bullet & \bullet & \bullet & \bullet & \bullet & 6 & \bullet & \bullet & \bullet & \\
& & & & & & 7 & & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 8 & \bullet & \bullet \\
& & & \bullet & \bullet & \bullet & \bullet & \bullet & 9 & \bullet \\
& & \bullet & & & & \bullet & \bullet & \bullet & 10
\end{pmatrix}}$$

Figure 5.10: A sample matrix and its LU factors. Diagonal elements a_{55} and a_{88} are zero.

$$\begin{array}{r}
1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10
\end{array}
\begin{pmatrix}
s_1 & s_1 & u_3 & & & u_6 & & & & \\
s_1 & s_1 & u_3 & u_4 & & u_6 & & u_8 & & \\
& & s_2 & & & & u_7 & u_8 & & \\
& & & s_3 & s_3 & s_3 & & & u_9 & \\
& & & s_3 & s_3 & s_3 & u_7 & & u_9 & \\
s_1 & s_1 & s_2 & s_3 & s_3 & s_3 & u_7 & u_8 & u_9 & \\
& & & & & & s_4 & & s_4 & s_4 \\
s_1 & s_1 & s_2 & s_3 & s_3 & s_3 & s_4 & s_4 & s_4 & s_4 \\
& & & s_3 & s_3 & s_3 & s_4 & s_4 & s_4 & s_4 \\
& & s_2 & & & & s_4 & s_4 & s_4 & s_4
\end{pmatrix}$$

Figure 5.11: Supernodal structure $bydefinitionT2$ of the factors of the sample matrix.



```
for column $j = 1$ to $n$ do
    $f = A(:, j)$;
    Symbolic factorization: determine which supernodes of $L$ will update $f$;
    Determine whether $j$ belongs to the same supernode as $j - 1$;
    for each updating supernode $(r{:}s) < j$ in topological order do
        Apply supernode-column update to column $j$:
            $f(r{:}s) = L(r{:}s, r{:}s)^{-1} \cdot f(r{:}s)$;  /* Now $f(r{:}s) = U(r{:}s, j)$ */
            $f(s+1{:}n) = f(s+1{:}n) - L(s+1{:}n, r{:}s) \cdot f(r{:}s)$;
    end for;
    Pivot: interchange $f(j)$ and $f(m)$, where $|f(m)| = \max |f(j{:}n)|$;
    Separate $L$ and $U$: $U(1{:}j, j) = f(1{:}j)$;   $L(j{:}n, j) = f(j{:}n)$;
    Scale: $L(j{:}n, j) = L(j{:}n, j) / L(j, j)$;
    Prune symbolic structure based on column $j$;
end for;
```

Figure 5.12: LU factorization with supernode-column updates

In symmetric Cholesky factorization, one type of supernodes - the ”fundamental” supernodes - can be made contiguous by permuting the matrix (symmetrically) according to a postorder on its elimination tree. This postorder is an example of what Liu calls an equivalent reordering, which does not change the sparsity of the factor. The postordered elimination tree can also be used to locate the supernodes before the numeric factorization.

We proceed similarly for the unsymmetric case. Here the appropriate analogue of the symmetric elimination tree is the *column elimination tree*, or column etree for short. The vertices of this tree are the integers 1 through n, representing the columns of A. The column etree of A is the (symmetric) elimination tree of the column intersection graph of A, or equivalently the elimination tree of $A^T A$ provided there is no cancellation in computing $A^T A$. See Gilbert and Ng for complete definitions. The column etree can be computed from A in time almost linear in the number of nonzeros of A.

Just as a postorder on the symmetric elimination tree brings together symmetric supernodes, we expect a postorder on the column etree to bring together unsymmetric supernodes. Thus, before we factor the matrix, we compute its column etree and permute the matrix columns according to a postorder on the tree.

5.7.3 Relaxed Supernodes

For most matrices, the average size of a supernode is only about 2 to 3 columns (though a few supernodes are much larger). A large percentage of supernodes consist of only a single column, many of which are leaves of the column etree. Therefore merging groups of columns at the fringe of the etree into *artificial supernodes* regardless of their row structures can be beneficial. A parameter r controls the granularity of the merge. A good merge rule is: node i is merged with its parent node j when the subtree rooted at j has at most r nodes. In practice, the best values of r are generally between 4 and 8 and yield improvements in running time of 5% to 15%.

Artificial supernodes are a special case of relaxed supernodes. They allow a small number of zeros in the structure of any supernode, thus relaxing the condition that the columns must have strictly nested structures.

5.7.4 Supernodal Numeric Factorization

Now we show how to modify the col-col algorithm to use sup-col updates and supernode-panel updates. This section describes the numerical computation involved in the updates.

Supernode-Column Updated

Figure 5.12 sketches the sup-col algorithm. The only difference from the col-col algorithm is that all the updates to a column from a single supernode are done together. Consider a supernode $(r : s)$ that updates column j. The coefficients of the updates are the values from a segment of column j of U, namely $U(r : s, j)$. The nonzero structure of such a segment is particularly simple: all the nonzeros are contiguous, and follow all the zeros. Thus, if k is the index of the first nonzero row in $U(r : s, j)$, the updates to column j from supernode $(r : s)$ come from columns k through s. Since the supernode is stored as a dense matrix, these updates can be performed by a dense lower triangular solve (with the matrix $L(k : s, k : s)$) and a dense matrix-vector multiplication (with the matrix $L(s + 1 : n, k : s)$). The symbolic phase determines the value of k, that is, the position of the first nonzero in the segment $U(r : s, j)$.

The advantages of using sup-col updates are similar to those in the symmetric case. Efficient Level 2 BLAS matrix-vector kernels can be used for the triangular solve and matrix-vector multiply. Furthermore, all the updates from the supernodal columns can be collected in a dense vector before doing a single scatter into the target vector. This reduces the amount of indirect addressing.

Supernode-Panel Updates

We can improve the sup-col algorithm further on machines with a memory hierarchy by changing the data access pattern. The data we are accessing in the inner loop (lines 5-9 of Figure 5.12) include the destination column j and all the updating supernodes $(r : s)$ to the left of column j. Column j is accessed many times, while each supernode $(r : s)$ is used only once. In practice, the number of nonzero elements in column j is much less than that in the updating supernodes. Therefore, the access pattern given by this loop provides little opportunity to reuse cached data. In particular, the same supernode $(r : s)$ may be needed to update both columns j and $j + 1$. But when we factor the (j+1)th column (in the next iteration of the outer loop), we will have to fetch supernode $(r : s)$ again from memory, instead of from cache (unless the supernodes are small compared to the cache).

Panels

To exploit memory locality, we factor several columns (say w of them) at a time in the outer loop, so that one updating supernode $(r : s)$ can be used to update as many of the w columns as possible. We refer to these w consecutive columns as a *panel* to differentiate them from a supernode, the row structures of these columns may not be correlated in any fashion, and the boundaries between panels may be different from those between supernodes. The new method requires rewriting the doubly nested loop as the triple loop shown in Figure 5.13.

The structure of each sup-col update is the same as in the sup-col algorithm. For each supernode $(r : s)$ to the left of column j, if $u_{kj} \neq 0$ for some $r \leq k \leq s$, then $u_{ij} \neq 0$ for all $k \leq i \leq s$. Therefore, the nonzero structure of the panel of U consists of dense column segments that are row-wise separated by supernodal boundaries, as in Figure 5.13. Thus, it is sufficient for the symbolic factorization algorithm to record only the first nonzero position of each column segment.

```
for column $j = 1$ to $n$ step $w$ do
    Symbolic factor: determine which supernodes will update any of $L(:, j:j+w-1)$;
    for each updating supernode $(r:s) < j$ in topological order do
        for column $jj = j$ to $j+w-1$ do
            Apply supernode-column update to column $jj$;
        end for;
    end for;
    Inner factorization:
        Apply the sup-col algorithm on columns and supernodes within the panel;
end for;
```

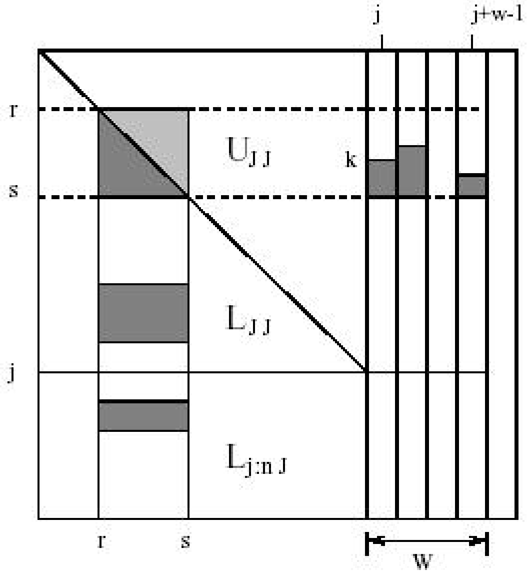


Figure 5.13: The supernode-panel algorithm, with columnwise blocking. $J = 1 : j - 1$

As detailed in section 4.4, symbolic factorization is applied to all the columns in a panel at once, over all the updating supernodes, before the numeric factorization step.

In dense factorization, the entire supernode-panel update in lines 3-7 of Figure 5.13 would be implemented as two Level 3 BLAS calls: a dense triangular solve with w right-hand sides, followed by a dense matrix-matrix multiply. In the sparse case, this is not possible, because the different sup-col updates begin at different positions k within the supernode, and the submatrix $U(r : s, j : j + w - 1)$ is not dense. Thus the sparse supernode-panel algorithm still calls the Level 2 BLAS. However, we get similar cache benefits to those from the Level 3 BLAS, at the cost of doing the loop reorganization ourselves. Thus we sometimes call the kernel of this algorithm a "BLAS-$2\frac{1}{2}$" method.

In the doubly nested loop (lines 3-7 of Figure 5.13), the ideal circumstance is that all w columns in the panel require updates from supernode $(r : s)$. Then this supernode will be used w times before it is forced out of the cache. There is a trade-off between the value of w and the size of the cache. For this scheme to work efficiently, we need to ensure that the nonzeros in the w columns do not cause cache thrashing. That is, we must keep w small enough so that all the data accessed in this doubly nested loop fit in cache. Otherwise, the cache interference between the source supernode and the destination panel can offset the benefit of the new algorithm.

5.8 Efficient sparse matrix algorithms

5.8.1 Scalable algorithms

By a *scalable* algorithm for a problem, we mean one that maintains efficiency bounded away from zero as the number p of processors grows and the size of the data structures grows roughly linearly in p.

Notable efforts at analysis of the scalability of dense matrix computations include those of Li and Coleman [58] for dense triangular systems, and Saad and Schultz [85]; Ostrouchov, *et al.* [73], and George, Liu, and Ng [39] have made some analyses for algorithms that map matrix columns to processors. Rothberg and Gupta [81] is a important paper for its analysis of the effect of caches on sparse matrix algorithms.

Consider any distributed-memory computation. In order to assess the communication costs analytically, it s useful to employ certain abstract lower bounds. Our model assumes that machine topology is given. It assumes that memory consists of the memories local to processors. It assumes that the communication channels are the edges of a given undirected graph $G = (W, L)$, and that processor–memory units are situated at some, possibly all, of the vertices of the graph. The model includes hypercube and grid-structured message-passing machines, shared-memory machines having physically distributed memory (the Tera machine) as well as tree-structured machines like a CM-5.

Let $V \subseteq W$ be the set of all processors and L be the set of all communication links.

We assume identical links. Let β be the inverse bandwidth (slowness) of a link in seconds per word. (We ignore latency in this model; most large distributed memory computations are bandwidth limited.)

We assume that processors are identical. Let ϕ be the inverse computation rate of a processor in seconds per floating-point operation. Let β_0 be the rate at which a processor can send or receive data, in seconds per word. We expect that β_0 and β will be roughly the same.

A distributed-memory computation consists of a set of processes that exchange information by sending and receiving messages. Let M be the set of all messages communicated. For $m \in M$,

$|m|$ denotes the number of words in m. Each message m has a source processor $src(m)$ and a destination processor $dest(m)$, both elements of V.

For $m \in M$, let $d(m)$ denote the length of the path taken by m from the source of the message m to its destination. We assume that each message takes a certain path of links from its source to its destination processor. Let $p(m) = (\ell_1, \ell_2, \ldots, \ell_{d(m)})$ be the path taken by message m. For any link $\ell \in L$, let the set of messages whose paths utilize ℓ, $\{m \in M \mid \ell \in p(m)\}$, be denoted $M(\ell)$.

The following are obviously lower bounds on the completion time of the computation. The first three bounds are computable from the set of message M, each of which is characterized by its size and its endpoints. The last depends on knowledge of the paths $p(M)$ taken by the messages.

1. (Average flux)

   $$\frac{\sum_{m\in M} |m| \cdot d(m)}{|L|} \cdot \beta.$$

   This is the total flux of data, measured in word-hops, divided by the machine's total communication bandwidth, L/β.

2. (Bisection width) Given $V_0, V_1 \subseteq W$, V_0 and V_1 disjoint, define

   $$sep(V_0, V_1) \equiv \min |\{L' \subseteq L \mid L' \text{ is an edge separator of } V_0 \text{ and } V_1\}|$$

   and

   $$flux(V_0, V_1) \equiv \sum_{\{m\in M \ \mid \ src(m)\in V_i, dest(m)\in V_{1-i}\}} |m| \,.$$

   The bound is

   $$\frac{flux(V_0, V_1)}{sep(V_0, V_1)} \cdot \beta.$$

   This is the number of words that cross from one part of the machine to the other, divided by the bandwidth of the wires that link them.

3. (Arrivals/Departures (also known as node congestion))

   $$\max_{v\in V} \sum_{dest(m) = v} |m|\beta_0;$$

   $$\max_{v\in V} \sum_{src(m) = v} |m|\beta_0.$$

   This is a lower bound on the communication time for the processor with the most traffic into or out of it.

4. (Edge contention)

   $$\max_{\ell\in L} \sum_{m\in M(\ell)} |m|\beta.$$

   This is a lower bound on the time needed by the most heavily used wire to handle all its traffic.

Of course, the actual communication time may be greater than any of the bounds. In particular, the communication resources (the wires in the machine) need to be scheduled. This can be done dynamically or, when the set of messages is known in advance, statically. With detailed knowledge of the schedule of use of the wires, better bounds can be obtained. For the purposes of analysis of algorithms and assignment of tasks to processors, however, we have found this more realistic approach to be unnecessarily cumbersome. We prefer to use the four bounds above, which depend only on the integrated (i.e. time-independent) information M and, in the case of the edge-contention bound, the paths $p(M)$. In fact, in the work below, we won't assume knowledge of paths and we won't use the edge contention bound.

5.8.2 Cholesky factorization

We'll use the techniques we've introduced to analyze alternative distributed memory implementations of a very important computation, Cholesky factorization of a symmetric, positive definite (SPD) matrix A. The factorization is $A = LL^T$ where L is lower triangular; A is given, L is to be computed.

The algorithm is this:

```
1.  $L := A$
2.  for $k = 1$ to $N$ do
3.      $L_{kk} := \sqrt{L_{kk}}$
4.      for $i = k + 1$ to $N$ do
5.          $L_{ik} := L_{ik} L_{kk}^{-1}$
6.      for $j = k + 1$ to $N$ do
7.          for $i = j$ to $N$ do
8.              $L_{ij} := L_{ij} - L_{ik} L_{jk}^T$
```

We can let the elements L_{ij} be scalars, in which case this is the usual or "point" Cholesky algorithm. Or we can take L_{ij} to be a block, obtained by dividing the rows into contiguous subsets and making the same decomposition of the columns, so that diagonal blocks are square. In the block case, the computation of $\sqrt{L_{kk}}$ (Step 3) returns the (point) Cholesky factor of the SPD block L_{kk}. If A is sparse (has mostly zero entries) then L will be sparse too, although less so than A. In that case, only the non-zero entries in the sparse factor L are stored, and the multiplication/division in lines 5 and 8 are omitted if they compute zeros.

Mapping columns

Assume that the columns of a dense symmetric matrix of order N are mapped to processors cyclically: column j is stored in processor $map(j) \equiv j \bmod p$. Consider communication costs on two-dimensional grid or toroidal machines. Suppose that p is a perfect square and that the machine is a $\sqrt{p} \times \sqrt{p}$ grid. Consider a mapping of the computation in which the operations in line 8 are performed by processor $map(j)$. After performing the operations in line 5, processor $map(k)$ must send column k to all processors $\{map(j) \mid j > k\}$.

Let us fix our attention on 2D grids. There are $L = 2p + O(1)$ links. A column can be broadcast from its source to all other processors through a spanning tree of the machine, a tree of total length p reaching all the processors. Every matrix element will therefore travel over $p - 1$ links, so the total information flux is $(1/2)N^2 p$ and the average flux bound is $(1/4)N^2 \beta$.

Type of Bound	Lower bound
Arrivals	$\frac{1}{4}N^2\beta_0$
Average flux	$\frac{1}{4}N^2\beta$

Table 5.1: Communication Costs for Column-Mapped Full Cholesky.

Type of Bound	Lower bound
Arrivals	$\frac{1}{4}N^2\beta\left(\frac{1}{p_r}+\frac{1}{p_c}\right)$
Edge contention	$N^2\beta\left(\frac{1}{p_r}+\frac{1}{p_c}\right)$

Table 5.2: Communication Costs for Torus-Mapped Full Cholesky.

Only $O(N^2/p)$ words leave any processor. If $N \gg p$, processors must accept almost the whole $(1/2)N^2$ words of L as arriving columns. The bandwidth per processor is β_0, so the arrivals bound is $(1/2)N^2\beta_0$ seconds. If $N \approx p$ the bound drops to half that, $(1/4)N^2\beta_0$ seconds. We summarize these bounds for 2D grids in Table 5.1.

We can immediately conclude that this is a nonscalable distributed algorithm. We may not take $p > \frac{N\phi}{\beta}$ and still achieve high efficiency.

Mapping blocks

Dongarra, Van de Geijn, and Walker [26] have shown that on the Intel Touchstone Delta machine ($p = 528$), mapping blocks is better than mapping columns in LU factorization. In such a mapping, we view the machine as an $p_r \times p_c$ grid and we map elements A_{ij} and L_{ij} to processor $(mapr(i), mapc(j))$. We assume a cyclic mappings here: $mapr(i) \equiv i \bmod p_r$ and similarly for $mapc$.

The analysis of the preceding section may now be done for this mapping. Results are summarized in Table 5.2. With p_r and p_c both $O(\sqrt{p})$, the communication time drops like $O(p^{-1/2})$. With this mapping, the algorithm is scalable even when $\beta \gg \phi$. Now, with $p = O(N^2)$, both the compute time and the communication lower bounds agree; they are $O(N)$. Therefore, we remain efficient when storage per processor is $O(1)$. (This scalable algorithm for distributed Cholesky is due to O'Leary and Stewart [72].)

5.8.3 Distributed sparse Cholesky and the model problem

In the sparse case, the same holds true. To see why this must be true, we need only observe that most of the work in sparse Cholesky factorization takes the form of the factorization of dense submatrices that form during the Cholesky algorithm. Rothberg and Gupta demonstrated this fact in their work in 1992 – 1994.

Unfortunately, with naive cyclic mappings, block-oriented approaches suffer from poor balance of the computational load and modest efficiency. Heuristic remapping of the block rows and columns can remove load imbalance as a cause of inefficiency.

Several researchers have obtained excellent performance using a block-oriented approach, both on fine-grained, massively-parallel SIMD machines [23] and on coarse-grained, highly-parallel MIMD machines [82]. A block mapping maps rectangular blocks of the sparse matrix to processors. A 2-D mapping views the machine as a 2-D $p_r \times p_c$ processor grid, whose members are denoted $p(i, j)$. To date, the 2-D cyclic (also called torus-wrap) mapping has been used: block L_{ij} resides at processor $p(i \bmod p_r, j \bmod p_c)$. All blocks in a given block row are mapped to the same row of processors, and all elements of a block column to a single processor column. Communication volumes grow as the square root of the number of processors, versus linearly for the 1-D mapping; 2-D mappings also asymptotically reduce the critical path length. These advantages accrue even when the underlying machine has some interconnection network whose topology is not a grid.

A 2-D cyclic mapping, however, produces significant load imbalance that severely limits achieved efficiency. On systems (such as the Intel Paragon) with high interprocessor communication bandwidth this load imbalance limits efficiency to a greater degree than communication or want of parallelism.

An alternative, heuristic 2-D block mapping succeeds in reducing load imbalance to a point where it is no longer the most serious bottleneck in the computation. On the Intel Paragon the block mapping heuristic produces a roughly 20% increase in performance compared with the cyclic mapping.

In addition, a scheduling strategy for determining the order in which available tasks are performed adds another 10% improvement.

5.8.4 Parallel Block-Oriented Sparse Cholesky Factorization

In the block factorization approach considered here, matrix blocks are formed by dividing the columns of the $n \times n$ matrix into N contiguous subsets, $N \leq n$. The identical partitioning is performed on the rows. A block L_{ij} in the sparse matrix is formed from the elements that fall simultaneously in row subset i and column subset j.

Each block L_{ij} has an owning processor. The owner of L_{ij} performs all block operations that update L_{ij} (this is the "owner-computes" rule for assigning work). Interprocessor communication is required whenever a block on one processor updates a block on another processor.

Assume that the processors can be arranged as a grid of p_r rows and p_c columns. In a *Cartesian product* (CP) mapping, $map(i, j) = p(RowMap(i), ColMap(j))$, where $RowMap : \{0..N-1\} \to \{0..p_r - 1\}$, and $ColMap : \{0..N-1\} \to \{0..p_c - 1\}$ are given mappings of rows and columns to processor rows and columns. We say that *map* is *symmetric Cartesian* (SC) if $p_r = p_c$ and $RowMap = ColMap$. The usual 2-D cyclic mapping is SC.[2]

5.9 Load balance with cyclic mapping

Any CP mapping is effective at reducing communication. While the 2-D cyclic mapping is CP, unfortunately it is not very effective at balancing computational load. Experiment and analysis show that the cyclic mapping produces particularly poor load balance; moreover, some serious load balance difficulties must occur for any SC mapping. Improvements obtained by the use of nonsymmetric CP mappings are discussed in the following section.

[2] See [82] for a discussion of *domains*, portions of the matrix mapped in a 1-D manner to further reduce communication.

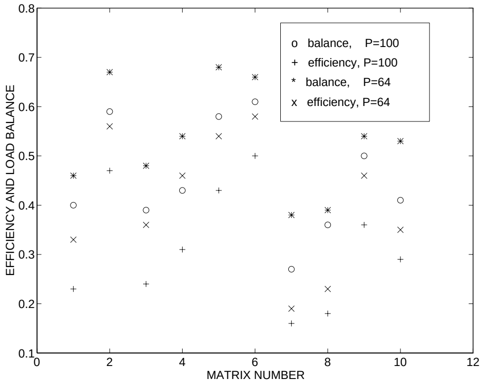


Figure 5.14: Efficiency and overall balance on the Paragon system ($B = 48$).

Our experiments employ a set of test matrices including two dense matrices (DENSE1024 and DENSE2048), two 2-D grid problems (GRID150 and GRID300), two 3-D grid problems (CUBE30 and CUBE35), and 4 irregular sparse matrices from the Harwell-Boeing sparse matrix test set [27]. Nested dissection or minimum degree orderings are used. In all our experiments, we choose $p_r = p_c = \sqrt{P}$, and we use a block size of 48. All Mflops measurements presented here are computed by dividing the operation counts of the best known sequential algorithm by parallel runtimes. Our experiments were performed on an Intel Paragon, using hand-optimized versions of the Level-3 BLAS for almost all arithmetic.

5.9.1 Empirical Load Balance Results

We now report on the efficiency and load balance of the method. Parallel efficiency is given by $t_{seq}/(P \cdot t_{par})$, where t_{par} is the parallel runtime, P is the number of processors, and t_{seq} is the runtime for the same problem on one processor. For the data we report here, we measured t_{seq} by factoring the benchmark matrices using our parallel algorithm on one processor. The overall balance of a distributed computation is given by $work_{total}/(P \cdot work_{max})$, where $work_{total}$ is the total amount of work performed in the factorization, P is the number of processors, and $work_{max}$ is the maximum amount of work assigned to any processor. Clearly, overall balance is an upper bound on efficiency.

Figure 1 shows efficiency and overall balance with the cyclic mapping. Observe that load balance and efficiency are generally quite low, and that they are well correlated. Clearly, load balance alone is not a perfect predictor of efficiency. Other factors limit performance. Examples include interprocessor communication costs, which we measured at 5% — 20% of total runtime, long critical paths, which can limit the number of block operations that can be performed concurrently, and poor scheduling, which can cause processors to wait for block operations on other processors to complete. Despite these disparities, the data indicate that load imbalance is an important contributor to reduced efficiency.

We next measured load imbalance among rows of processors, columns of processors, and diagonals of processors. Define $work[i, j]$ to be the runtime due to updating of block L_{ij} by its owner. To approximate runtime, we use an empirically calibrated estimate of the form work = operations + $\omega \cdot$ block-operations; on the Paragon, $\omega = 1,000$.

Define $RowWork[i]$ to be the aggregate work required by blocks in row i: $RowWork[i] = \sum_{j=0}^{N-1} work[i, j]$. An analogous definition applies for $ColWork$, the aggregate column work. Define

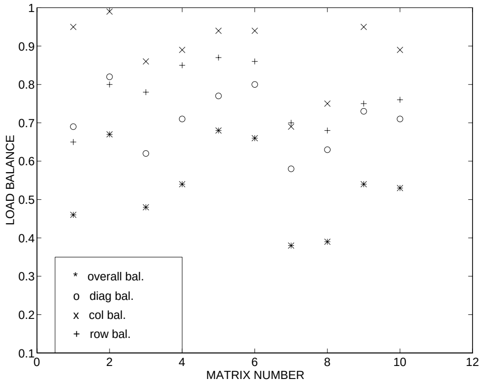


Figure 5.15: Efficiency bounds for 2-D cyclic mapping due to row, column and diagonal imbalances ($P = 64, B = 48$).

row balance by $work_{total}/p_r \cdot work_{rowmax}$, where $work_{rowmax} = \max_r \sum_{i:RowMap[i]=r} RowWork[i]$. This row balance statistic gives the best possible overall balance (and hence efficiency), obtained only if there is perfect load balance within each processor row. It isolates load imbalance due to an overloaded processor row caused by a poor row mapping. An analogous expression gives column balance, and a third analogous expression gives diagonal balance. (Diagonal d is made up of the set of processors $p(i, j)$ for which $(i - j) \bmod p_r = d$.) While these three aggregate measures of load balance are only upper bounds on overall balance, the data we present later make it clear that improving these three measures of balance will in general improve the overall load balance.

Figure 2 shows the row, column, and diagonal balances with a 2-D cyclic mapping of the benchmark matrices on 64 processors. Diagonal imbalance is the most severe, followed by row imbalance, followed by column imbalance.

These data can be better understood by considering dense matrices as examples (although the following observations apply to a considerable degree to sparse matrices as well). Row imbalance is due mainly to the fact that $RowWork[i]$, the amount of work associated with a row of blocks, increases with increasing i. More precisely, since $work[i, j]$ increases linearly with j and the number of blocks in a row increases linearly with i, it follows that $RowWork[i]$ increases quadratically in i. Thus, the processor row that receives the last block row in the matrix receives significantly more work than the processor row immediately following it in the cyclic ordering, resulting in significant row imbalance. Column imbalance is not nearly as severe as row imbalance. The reason, we believe, is that while the work associated with blocks in a column increases linearly with the column number j, the number of blocks in the column *decreases* linearly with j. As a result, $ColWork[j]$ is neither strictly increasing nor strictly decreasing. In the experiments, row balance is indeed poorer than column balance. Note that the reason for the row and column imbalance is not that the 2-D cyclic mapping is an SC mapping; rather, we have significant imbalance because the mapping functions $RowMap$ and $ColMap$ are each poorly chosen.

To better understand diagonal imbalance, one should note that blocks on the diagonal of the matrix are mapped exclusively to processors on the main diagonal of the processor grid. Blocks just below the diagonal are mapped exclusively to processors just below the main diagonal of the processor grid. These diagonal and sub-diagonal blocks are among the most work-intensive blocks in the matrix. In sparse problems, moreover, the diagonal blocks are the only ones that are guaranteed to be dense. (For the two dense test matrices, diagonal balance is not significantly worse than row balance.) The remarks we make about diagonal blocks and diagonal processors apply to *any* SC mapping, and do not depend on the use of a cyclic function $RowMap(i) = i \bmod p_r$.

5.10 Heuristic Remapping

Nonsymmetric CP mappings, which map rows independently of columns, are a way to avoid diagonal imbalance that is automatic with SC mappings. We shall choose the row mapping $RowMap$ to maximize the row balance, and independently choose $ColMap$ to maximize column balance. Since the row mapping has no effect on the column balance, and vice versa, we may choose the row mapping in order to maximize row balance independent of the choice of column mapping.

The problems of determining $RowMap$ and $ColMap$ are each cases of a standard NP-complete problem, *number partitioning* [36], for which a simple heuristic is known to be good [3]. This heuristic obtains a row mapping by considering the block rows in some predefined sequence. For each processor row, it maintains the total work for all blocks mapped to that row. The algorithm iterates over block rows, mapping a block row to the processor row that has received the least work thus far. We have experimented with several different sequences, the two best of which we now describe.

The **Decreasing Work (DW)** heuristic considers rows in order of decreasing work. This is a standard approach to number partitioning; that small values toward the end of the sequence allow the algorithm to lessen any imbalance caused by large values encountered early in the sequence.

The **Increasing Depth (ID)** heuristic considers rows in order of increasing depth in the elimination tree. In a sparse problem, the work associated with a row is closely related to its depth in the elimination tree.

The effect of these schemes is dramatic. If we look at the three aggregate measures of load balance, we find that these heuristics produce row and column balance of 0.98 or better, and diagonal balance of 0.93 or better, for test case BCSSTK31, which is typical. With ID as the row mapping we have produced better than a 50% improvement in overall balance, and better than a 20% improvement in performance, on average over our test matrices, with $P = 100$. The DW heuristic produces only slightly less impressive improvements The choice of column mapping, as expected, is less important. In fact, for our test suite, the cyclic column mapping and ID row mapping gave the best mean performance. [4]

We also applied these ideas to four larger problems: DENSE4096, CUBE40, COPTER2 (a helicopter rotor blade model, from NASA) and 10FLEET (a linear programming formulation of an airline fleet assignment problem, from Delta Airlines). On 144 and 196 processors the heuristic (increasing depth on rows and cyclic on columns) again produces a roughly 20% performance improvement over a cyclic mapping. Peak performance of 2.3 Gflops for COPTER2 and 2.7 Gflops for 10FLEET were achieved; for the model problems CUBE40 and DENSE4096 the speeds were 3.2 and 5.2 Gflops.

In addition to the heuristics described so far, we also experimented with two other approaches to improving factorization load balance. The first is a subtle modification of the original heuristic. It begins by choosing some column mapping (we use a cyclic mapping). This approach then iterates over rows of blocks, mapping a row of blocks to a row of processors so as to minimize the amount of work assigned to any one *processor*. Recall that the earlier heuristic attempted to minimize the aggregate work assigned to an entire row of processors. We found that this alternative heuristic produced further large improvements in overall balance (typically 10-15% better than that of our

[3] Number partitioning is a well studied NP-complete problem. The objective is to distribute a set of numbers among a fixed number of bins so that the maximum sum in any bin is minimized.

[4] Full experimental data has appeared in another paper [83].

original heuristic). Unfortunately, realized performance did not improve. This result indicates that load balance is not the most important performance bottleneck once our original heuristic is applied.

A very simple alternate approach reduces imbalance by performing cyclic row and column mappings on a processor grid whose dimensions p_c and p_r are relatively prime; this reduces diagonal imbalance. We tried this using 7×9 and 9×11 processor grids (using one fewer processor that for our earlier experiments with $P = 64$ and $P = 100$.) The improvement in performance is somewhat lower than that achieved with our earlier remapping heuristic (17% and 18% mean improvement on 63 and 99 processors versus 20% and 24% on 64 and 100 processors). On the other hand, the mapping needn't be computed.

5.11 Scheduling Local Computations

The next questions to be addressed, clearly, are: (i) what is the most constraining bottleneck after our heuristic is applied, and (ii) can this bottleneck be addressed to further improve performance?

One potential remaining bottleneck is communication. Instrumentation of our block factorization code reveals that on the Paragon system, communication costs account for less than 20% of total runtime for all problems, even on 196 processors. The same instrumentation reveals that most of the processor time not spent performing useful factorization work is spent idle, waiting for the arrival of data.

We do not believe that the idle time is due to insufficient parallelism. Critical path analysis for problem BCSSTK15 on 100 processors, for example, indicates that it should be possible to obtain nearly 50% higher performance than we are currently obtaining. The same analysis for problem BCSSTK31 on 100 processors indicates that it should be possible to obtain roughly 30% higher performance. We therefore suspected that the scheduling of tasks by our code was not optimal.

To that end we tried alternative scheduling policies. They are:

FIFO. Tasks are initiated in the order in which the processor discovers that they are ready.

Destination depth. Ready tasks initiated in order of the destination block's elimination tree depth.

Source depth. Ready tasks initiated in order of the source block's elimination tree depth.

For the FIFO policy, a queue of ready tasks is used, while for the others, a heap is used. We experimented with 64 processors, using BSCCST31, BCSSTK33, and DENSE2048. Both priority-based schemes are better than FIFO; destination depth seems slightly better than source depth. We observed a slowdown of 2% due to the heap data structure on BCSSTK33; the destination priority scheme then improved performance by 15% for a net gain of 13%. For BSCCST31 the net gain was 8%. For DENSE2048, however, there was no gain. This improvement is encouraging. There may be more that can be achieved through the pursuit of a better understanding of the scheduling question.